University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Pitt SCI 5 YEARS

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 11: this Friday @ 11:59 pm
  - Lab 9: Tuesday 4/4 @ 11:59 pm
  - Assignment 4: Friday 4/14 @ 11:59 pm
    - Support video and slides on Canvas + Solution for Labs 8 and 9

Previous lecture

- Priority Queue ADT
  - Heap implementation

This Lecture

- Minimum Spanning Tree (MST)
  - Prim’s MST algorithm
    - naiive implementation
    - Best Edges array implementation
    - using a min-heap
  - Kruskal’s MST algorithm
- Weighted Shortest Paths problem
  - Dijkstra’s single-source shortest paths algorithm
  - Bellman-Ford’s shortest paths algorithm

Neighborhood connectivity Problem

- We want to keep a set of neighborhoods **connected** with the **minimum cost** possible
- **Input:** A set of neighborhoods and a file:
  - neighborhood i, neighborhood j, cost of connecting the two neighborhoods
  - …
- **Output:** A set of neighborhood pairs to be connected and a total cost such that
  - We can go from any neighborhood to any other **(connected)**
  - The total cost should be minimum (i.e., as small as it can be) **(minimal cost)**

Think Data Structures First!

- How can we structure the input in computer memory?
- Can we use Graphs?
- What about the costs? How can we model that?

We said spatial layouts of graphs were irrelevant

- We define graphs as sets of vertices and edges
- However, we'll certainly want to be able to reason about **bandwidth, distance, capacity**, etc. of the real world things our graph represents
  - Whether a link is 1 gigabit or 10 megabit will drastically affect our analysis of traffic flowing through a network
  - Whether a road is single-lane or 4-lane
  - Whether two airports are 2000 miles apart or 200 miles apart will have an effect on the number of flights going in and out between them

We can represent such information with edge weights

- How do we store edge weights?
  - Adjacency matrix?
  - Adjacency list?
  - Do we need a whole new graph representation?

New problems!

How do weights affect finding spanning trees and shortest paths?

- The weighted variants of these problems are called ***minimum spanning tree*** and ***weighted shortest path***

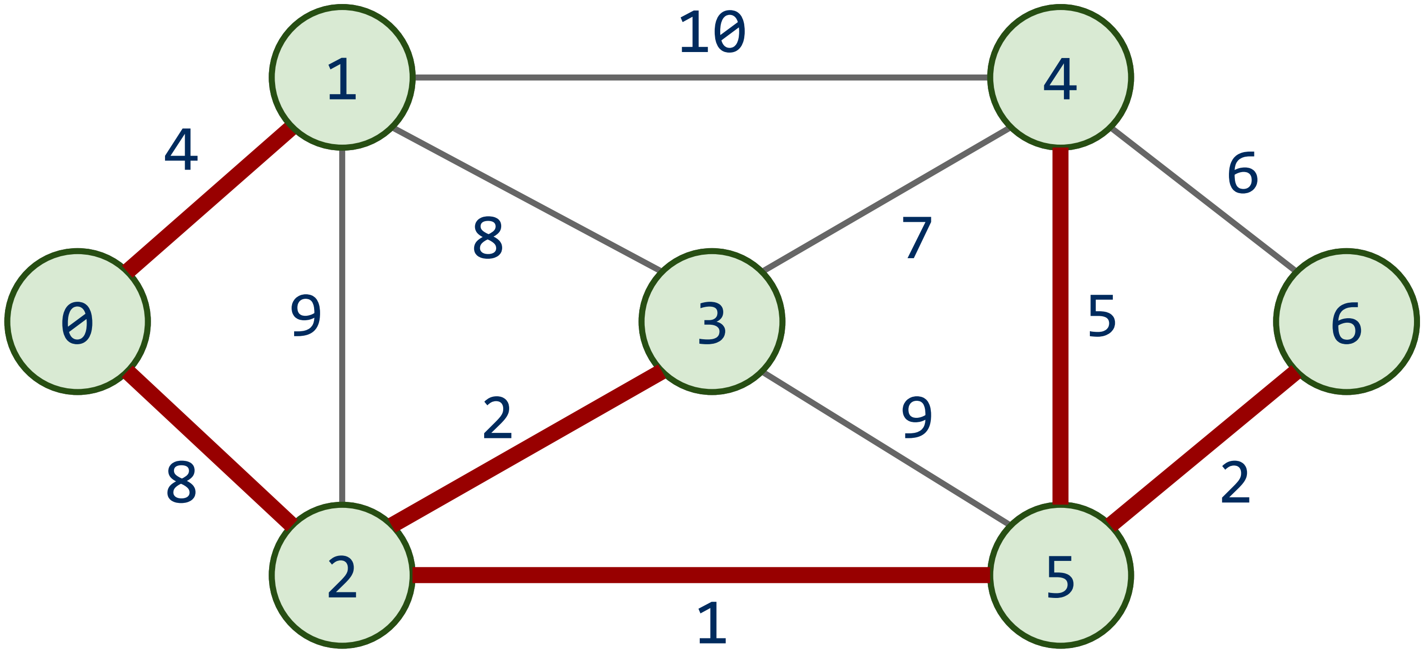
Minimum spanning trees (MST)

- Graphs can potentially have multiple spanning trees
    - BFS, DFS traversals find possible different spanning trees
- MST is the spanning tree that has the **minimum sum** of the weights of its edges

Prim’s algorithm

- Initialize T to contain the starting vertex
  - T will eventually become the MST
- While there are vertices not in T:
  - Find a **minimum edge-weight edge** that connects a vertex **in T** to a vertex **not yet in T**
  - Add the edge with its vertex to T

Prim’s algorithm



Runtime of Prim's

- At each step, check all possible edges
- For a complete graph:
  - First iteration:
    - v - 1 possible edges
  - Next iteration:
    - 2(v - 2) possibilities
      - Each vertex in T shared v-1 edges with other vertices, but the edges they shared with each other already in T
  - Next:
    - 3(v - 3) possibilities
  - …
- Runtime:
  - $\Sigma_{i\ =\ 1\ \text{to}\ v-1}$ (i * (v - i)) = Θ(largest term * number of terms)
  - number of terms = *v-1*
  - largest term is $v^2/4$ (when *i=v/2)*
  - Evaluates to $\Theta(v^3)$

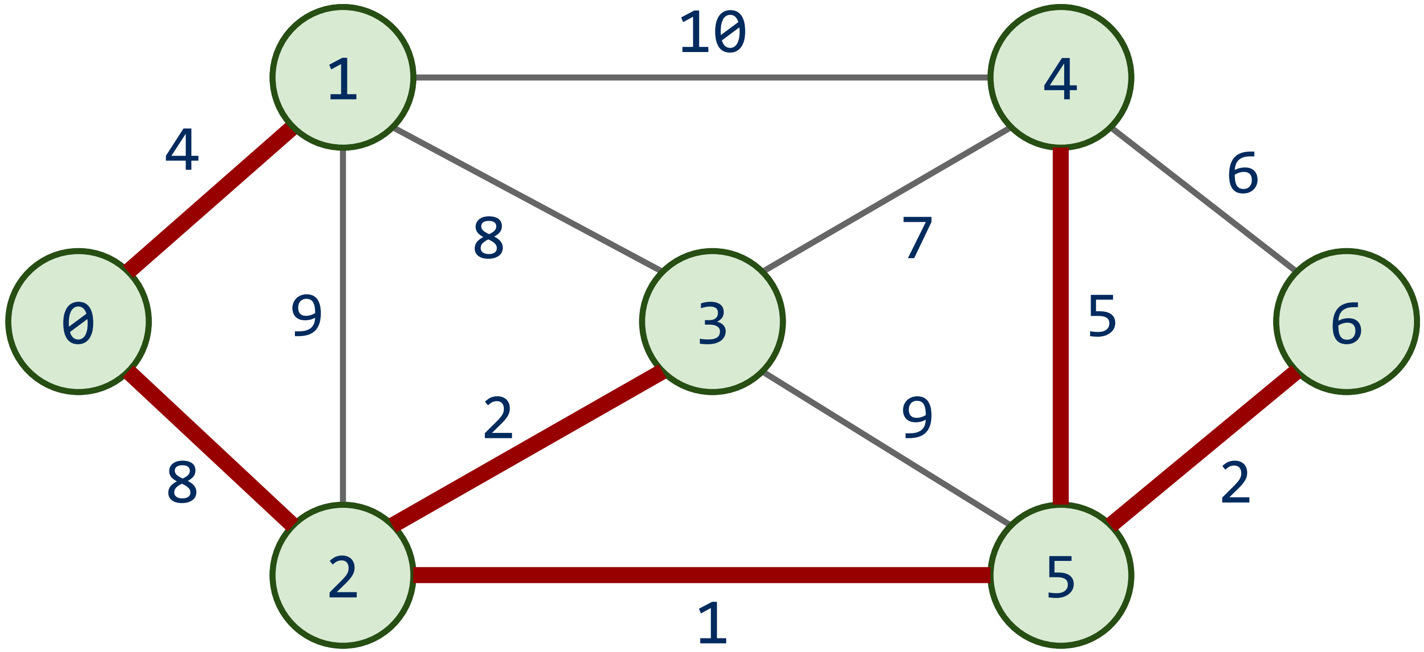
Do we need to look through all remaining edges?

- No! We only need to consider the ***best edge*** possible for each vertex!
- The best edge of a vertex is the edge with the **minimum weight** connecting to the vertex from a vertex already in T
  - Adding a vertex to T → new edges become
  - Best edge values can be **updated** as we add vertices to T

An enhanced implementation of Prim's Algorithm

- Add start vertex to T
- Repeat until all vertices added to T
  - Check the **neighbors of the added** vertex
    - update best edge values if needed
    - update **parent** as well
  - Add to T a vertex with the **smallest best edge**

Prim’s algorithm



	0	1	2	3	4	5	6
Parent:	--	0	0	2	5	2	5
Best Edge:	0	4	8	2	5	1	2

Runtime of the Best Edges Implementation

- For every vertex we add to T, check and possibly update neighbors
- Let's assume we use an **adjacency matrix**:
  - Takes Θ(v) to check the neighbors of a given vertex
  - Time to update parent/best edge arrays?
    - Θ(1)
  - Time to pick next vertex?
    - Θ(v)
- Total: $v * 2\,\Theta(v) = \Theta(v^2)$

Runtime of the Best Edges Implementation

- For every vertex we add to T, check and possibly update neighbors
- Let's assume we use an **adjacency lists**:
  - Takes $\Theta(d)$ to check the neighbors of a given vertex
  - Time to update parent/best edge arrays?
    - $\Theta(1)$
  - Time to pick next vertex?
    - $\Theta(v)$
- Total: $v * 2\ \Theta(v) = \Theta(v^2)$

Prim's MST Algorithm

- seen, parent, and BestEdge are arrays of size v
- Initialize seen to false, parent to -1, and BestEdge to infinity
- BestEdge[start] = 0
- for i = 0 to v-1
    - Find w s.t. seen[w] = false and BestEdge[w] is minimum over all unseen vertices
    - seen[w] = 1
    - for each neighbor x of w
        - if(BestEdge[x] > edge weight of edge (w, x)
            - BestEdge[x] = edge weight of (w, x)
            - parent[x] = w
- The parent array represents the found MST

Prim's MST Algorithm

- seen, parent, and BestEdge are arrays of size v
- Initialize seen to false, parent to -1, and BestEdge to infinity
- BestEdge[start] = 0
- for i = 0 to v-1
  - **Find w s.t. seen[w] = false and BestEdge[w] is minimum over all unseen vertices**
  - seen[w] = 1
  - for each neighbor x of w
    - if(BestEdge[x] > edge weight of edge (w, x)
      - BestEdge[x] = edge weight of (w, x)
      - parent[x] = w
- The parent array represents the found MST

What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
  - Priority queues can remove the min value stored in them in $\Theta(\log n)$
    - Also $\Theta(\log n)$ to add to the priority queue

Let's maintain best edge values in a PQ!

- PQ will need to be **indexable** to **update** the best edge
- This is the idea of ***eager Prim's***

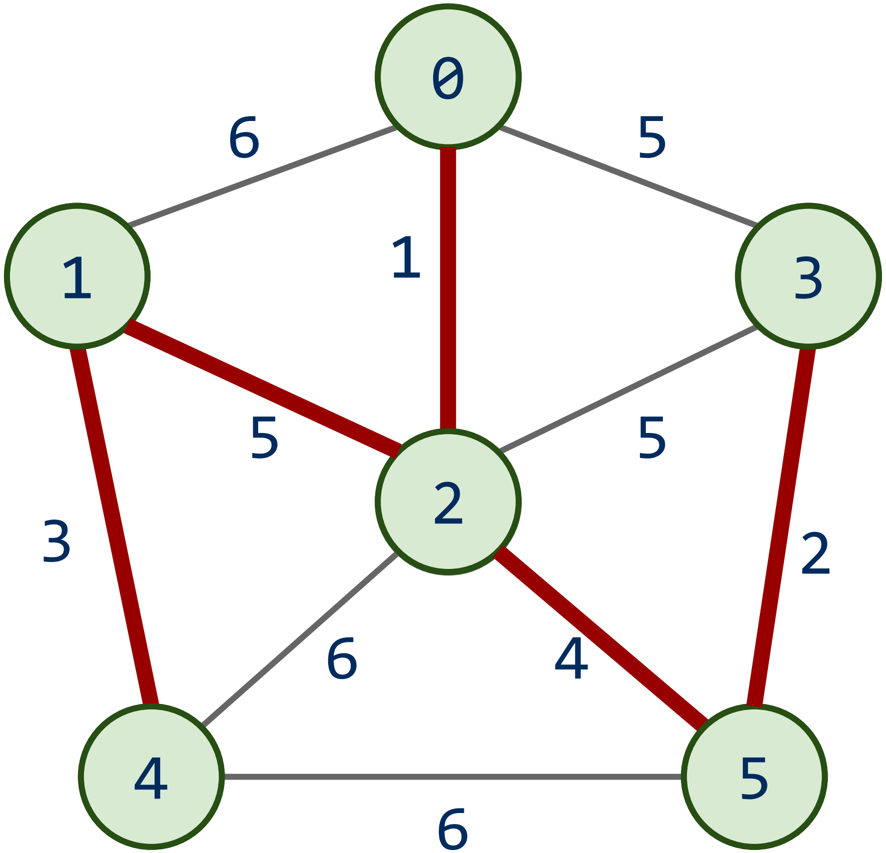
Eager Prim's Runtime

- v inserts
  - v log v
- e updates
  - e log v
- v removeMin
  - v log v
- Total: (e+v) log v
- Assuming connected graph
  - e >= v – 1
- e+v = Theta(e)
- Total runtime = e log v

We can be a bit lazy: let's keep e edges in the PQ

- PQ doesn't have to indexable
- Lazy Prim's implementation
    - Visit a vertex
    - Add edges coming out of it to a PQ
    - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat

Prim's with a priority queue



PQ:

1: (0, 2)

2: (5, 3)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (2, 1)

6: (0, 1)

6: (2, 4)

6: (5, 4)

Runtime using a priority queue

- Have to insert all e edges into the priority queue
    - In the worst case, we'll also have to remove all e edges
- So we have:
    - $e * \Theta(\lg e) + e * \Theta(\lg e)$
    - $= \Theta(2 * e \lg e)$
    - $= \Theta(e \lg e)$
- This algorithm is known as *lazy Prim's*

Comparison of Prim's implementations

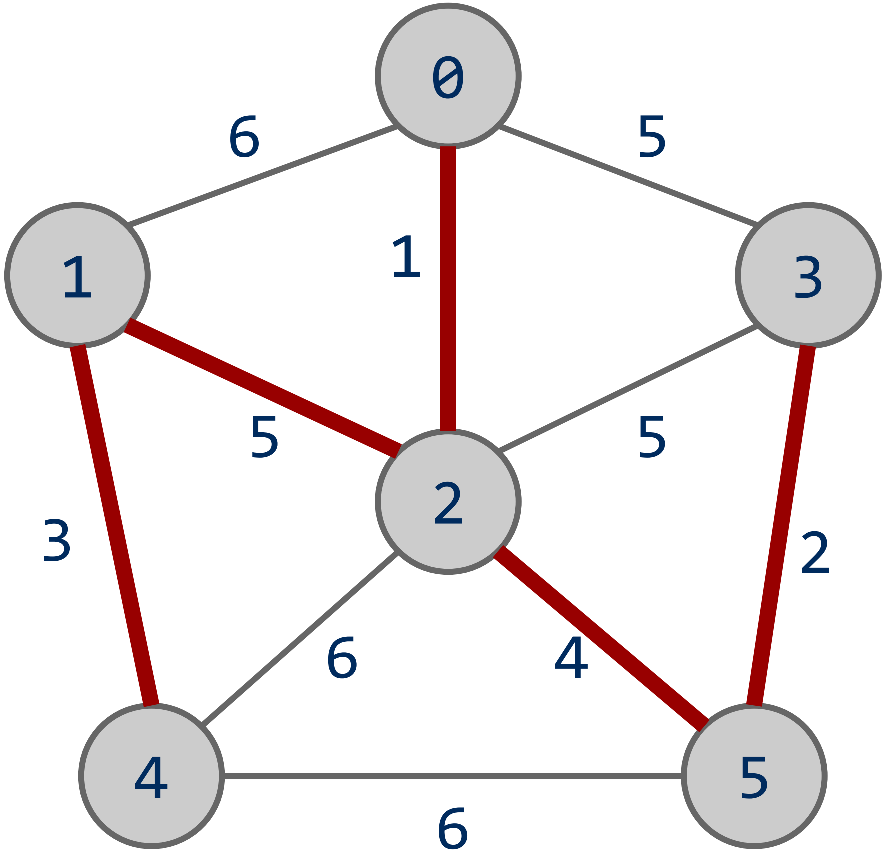
- Parent/Best Edge array Prim's
  - Runtime: $\Theta(v^2)$
  - Space: $\Theta(v)$
- Lazy Prim's
  - Runtime: $\Theta(e \lg e)$
  - Space: $\Theta(e)$
  - Requires a PQ
- Eager Prim's
  - Runtime: $\Theta(e \lg v)$
  - Space: $\Theta(v)$
  - Requires an indexable PQ

How do these compare?

Another MST algorithm

- Kruskal's MST:
  - Insert all edges into a PQ
  - Grab the min edge from the PQ that does not create a cycle in the MST
  - Remove it from the PQ and add it to the MST

Kruskal's example



PQ:

1: (0, 2)

2: (3, 5)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (1, 2)

6: (0, 1)

6: (2, 4)

6: (4, 5)

Kruskal’s runtime

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph
- How do we efficiently implement cycle detection?
  - BFS/DFS
    - v + e
  - Union/Find data structure
    - log v

Kruskal's Runtime

- e iterations
  - removeMin
    - log e
  - Cycle detection
    - v + e using DFS/BFS
    - log v using Union/Find
- Total: e log e
- Assuming connected graph
  - $v - 1 <= e <= v^2$
  - log v <= log e <= 2 log v
  - log e = Theta(log v)
- Total runtime: e log v
- Same as Prim's

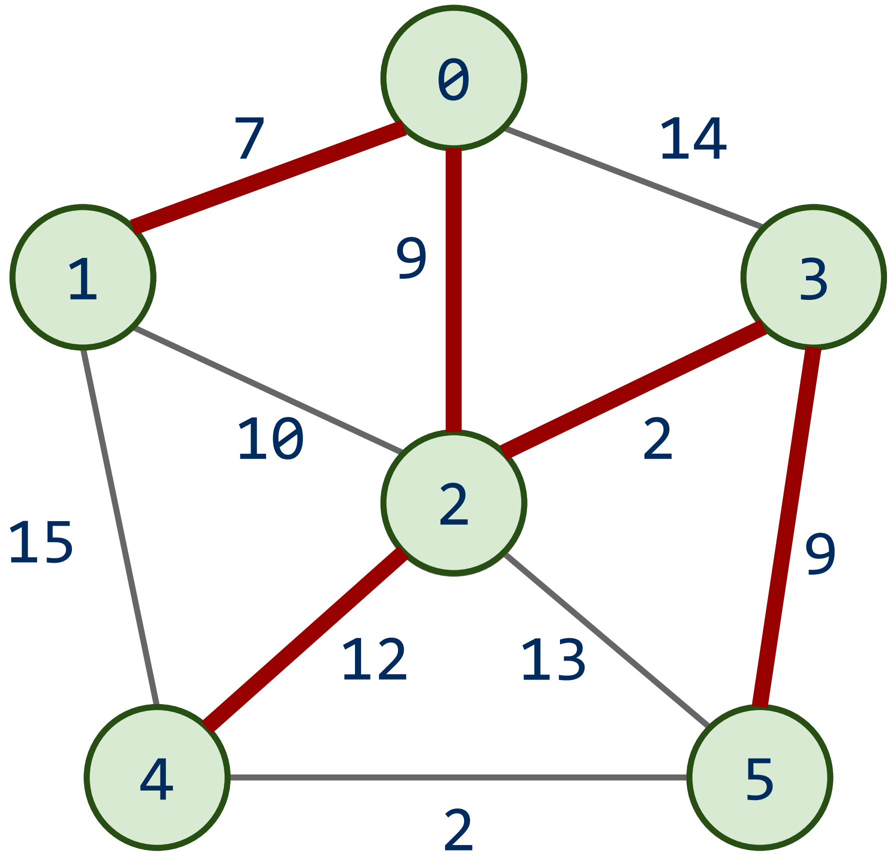
Problem of the Day: Weighted Shortest Paths

- Input:
  - A road network
    - Road segments and intersections
    - Road segments are labeled by travel time
      - From length and maximum speed
      - How do we get max speed?
  - Starting address and destination address
- Output:
  - A shortest path from source to destination

Dijkstra's algorithm

- Set a distance value of Double.POSITIVE_INFINITY for all vertices
- distance[start] = 0
- Set cur = start
- While destination is not visited:
  - For each unvisited neighbor x of cur:
    - Compute distance from start to x through cur
      - distance[cur] + weight of edge between cur and x
    - Update distance[x] if computed distance < distance[x]
  - Mark cur as visited
  - Let cur be the unvisited vertex with the smallest tentative distance from start

Dijkstra's example

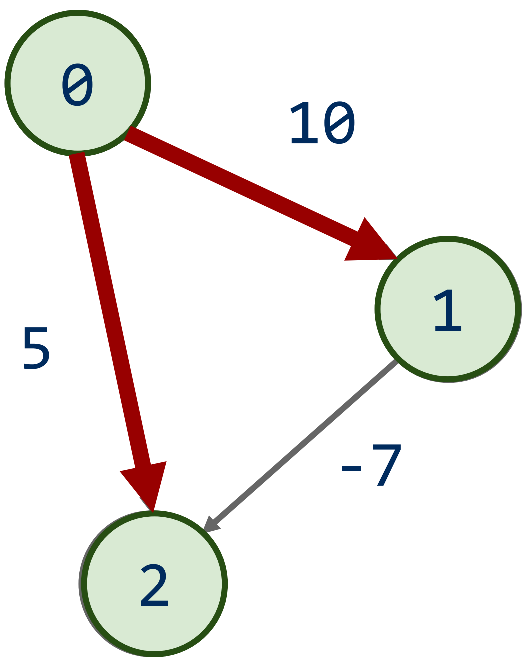


	Distance	Parent
0	0	--
1	7	0
2	9	0
3	11	2
4	21	2
5	20	3

Analysis of Dijkstra’s algorithm

- How to implement?
  - Best path/parent array?
    - Runtime?
  - PQ?
    - Turns out to be very similar to Eager Prims
      - Storing paths instead of edges
    - Runtime?

Dijkstra's example with negative edge weights



	Distance	Parent
0	0	--
1	10	0
2	5	0

Incorrect!

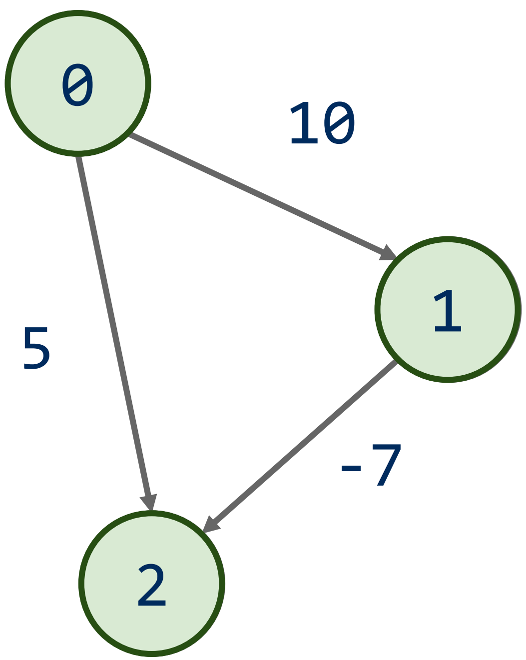
Analysis of Dijkstra’s algorithm

Dijkstra’s is correct only when all edge weights >= 0

Bellman-Ford's algorithm

- Set a distance value of Double.POSITIVE_INFINITY for all vertices
- Initialize a FIFO Q
- distance[start] = 0
- add start to Q
- While Q is not empty:
    - cur = pop a vertex from Q
    - For each non-parent neighbor x of cur:
        - Compute distance from start to x through cur
            - distance[cur] + weight of edge between cur and x
        - if computed distance < distance[x]
            - Update distance[x]
            - add x to Q if not already there

Bellman-Ford's example with negative edge weights



	Distance	Parent
0	0	--
1	10	0
2	3	1

FIFO Q:

0
1
2

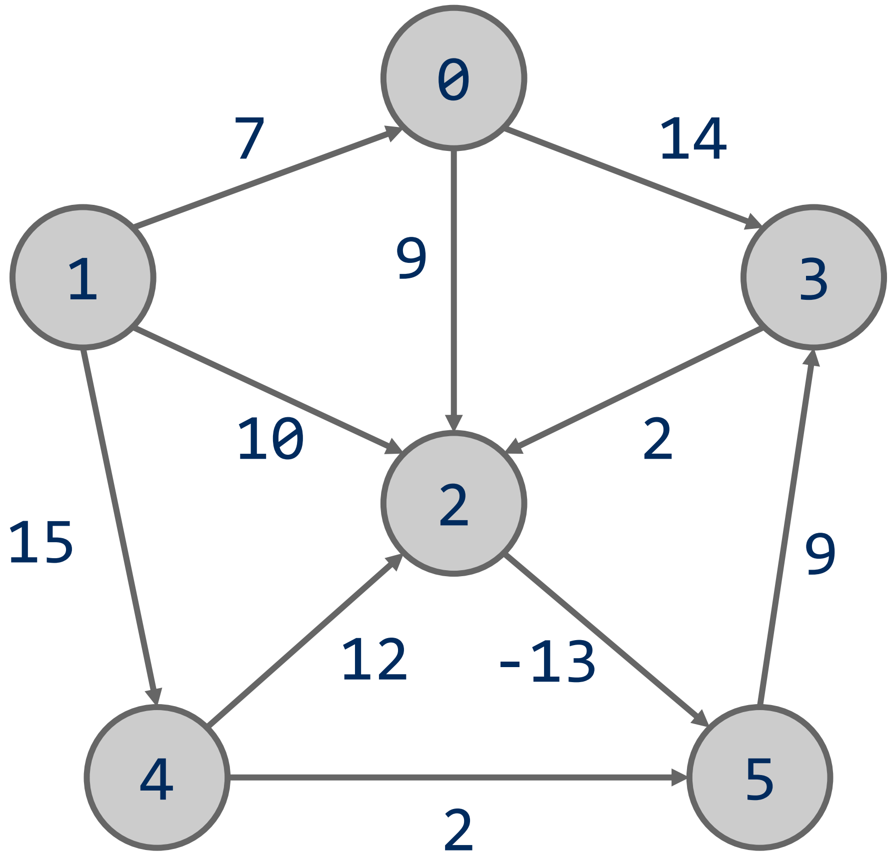
Correct!

Analysis of Bellman-Ford’s algorithm

Bellman-Ford’s is correct even when there are negative edge weights in the graph but what about negative cycles?

- a negative cycle is a cycle with a negative total weight

Bellman-Ford's example with a negative cycle



Bellman-Ford's algorithm

- Set a distance value of Double.POSITIVE_INFINITY for all vertices
- Initialize a FIFO Q
- distance[start] = 0
- add start to Q
- While Q is not empty **and no negative cycle has been detected**:
  - cur = pop a vertex from Q
  - For each non-parent neighbor x of cur:
    - Compute distance from start to x through cur
      - distance[cur] + weight of edge between cur and x
    - if computed distance < distance[x]
      - Update distance[x]
      - add x to Q if not already there
  - check for a negative cycle in the current Spanning Tree every v edges